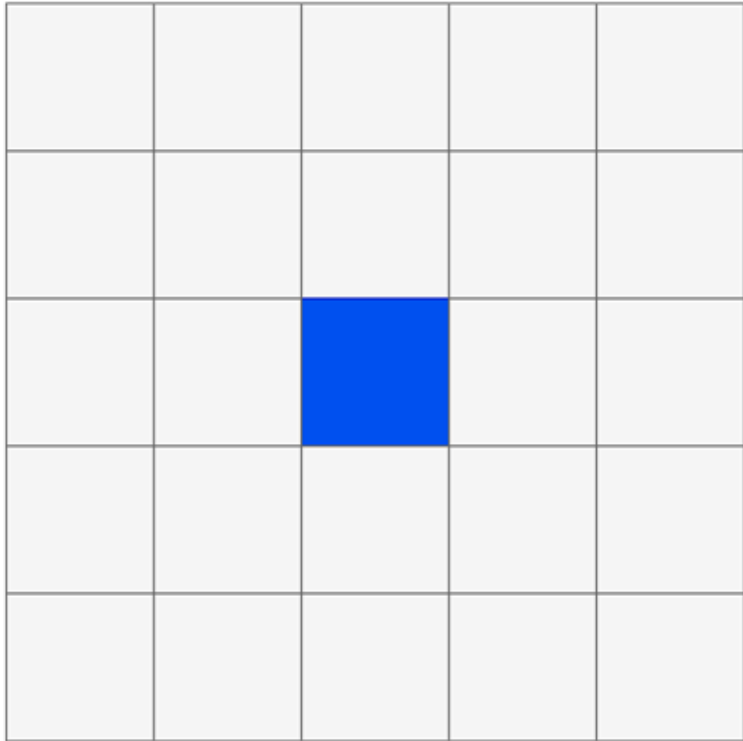


High Level Synthesis

2D cellular automata
custom hardware architecture

CA basics

A cellular automaton consists of a grid of cells. We call each consecutive state of the grid a generation.

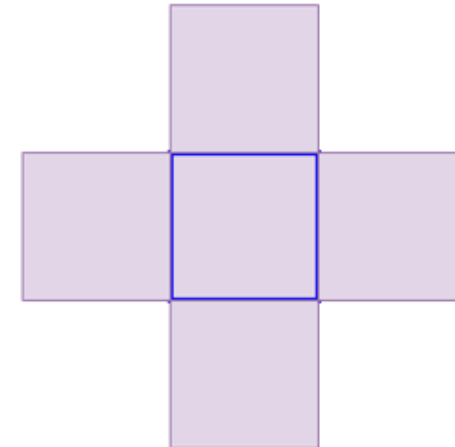


Possible cell values, for example $\{0,1\}$



Each cell can 'see' only its neighbors.
Neighborhood structure used:

Cross Kernel



CA basics

Each cell is updated on discrete time using a rule

We used outer totalistic rules:

The next value of a cell depends on the sum of its neighbors and the value of the cell itself.

Rules are specified with code names. For example, for the case of 2 possible cell values:

rule 224 = $(0011100000)_2$

The initial state of the grid is called **seed**

We set the middle cell to value 1 and the rest to 0

table showcasing the rule

center cell (t)	sum of neighbors	center cell (t+1)
0	0	0
1	0	0
0	1	0
1	1	0
0	2	0
1	2	1
0	3	1
1	3	1
0	4	0
1	4	0

A table like this is used as a LUT

C++ Implementation

We followed the parallel line buffers method for high throughput.

Initially we set:

- The dimensions of the input grid , NxM
- The amount of discrete values that each cell can hold
 - Calculates the bit precision needed
 - Calculates the required LUT dimensions, they depend on the possible cell values and the possible combinations of 4 neighbor cells sum

Also 2 testbench variables, the number of generations that the module should produce and whether the output should be verbose (print the tables and a message if they are correct) or not.

```
const int N = 25;
const int M = 25;

const int DISCRETE_VALUES = 4;
const int BITS = ac::log2_ceil<DISCRETE_VALUES>::val;
typedef ac_int<BITS, false> dtype;

const int LUT_DIM1 = DISCRETE_VALUES;
const int LUT_DIM2 = 4*(DISCRETE_VALUES-1) + 1;

const int GENERATIONS = 20;
bool VERBOSE = 0;
```

C++ Implementation

C++ class definition, RunCA, can be seen in the code files, here we show the main interface.

The module computes a generation on place, given an NxM grid and a LUT containing the update rule, using the standard implementation shown on class, repeating the cells at the borders of the grid.

Extra things that we tried but didn't quite work:

- Wrap around borders
 - Could be done but it would need an extra temporary buffer for the first row of the image which should be appended on the output image at the end of the execution, extra overhead defeats the purpose.
- Use smaller local_kernel, since we utilize only the cross area
 - Too much complexity for small gain of 1-2 registers, we just dropped the two leftmost, upper and lower, positions

```
#pragma hls_design interface
void compute_generation(dtype ca[N][M], dtype rule_lut[LUT_DIM1][LUT_DIM2]){
    for(int i=-1;i<N+1;i++){
        for(int j=-1;j<M+1;j++){
            local_kernel[1] = local_kernel[0];
            local_kernel[2] = local_kernel[3];
            local_kernel[0] = local_kernel[4];
            local_kernel[6] = local_kernel[5];

            local_kernel[3] = row_buffers[0][j+1];
            local_kernel[4] = row_buffers[1][j+1];

            row_buffers[0][j+1] = row_buffers[1][j+1];

            // handle invalid (for the image) indices
            if( (i<0) || (i>N-1) || (j<0) || (j>M-1) ){
                local_kernel[5] = row_buffers[1][j+1] = ca[clip(i,N-1)][clip(j,M-1)];
            }else{
                local_kernel[5] = row_buffers[1][j+1] = ca[i][j];
            }

            // compute the output after a given moment
            if((i>0 && i<=N) && (j>0 && j<=M)){
                ca[i-1][j-1] = apply_rule_custom(rule_lut);
            }
        }
    }
}
```

C++ runs

We have defined a separate class, TestCA, containing the module that generates a table that is used as ground truth. We implemented 2 testbench procedures.

ca_module_tb1.cc:

In the main function, we initialize the seed for the random generation, the 2 tables (cell grids), one that is updated by RunCA and another that is updated by TestCA, we set the seed of the grids, and run each module for the number defined by GENERATIONS. At each generation we provide a random generated LUT and with the help of added functions we evaluate the results.

Example of verbose run on the right, for 20 generations, $N=25$, $M=25$ and 4 discrete cell values, mapped on a set of symbols, $\{0,1,2,3\} \Rightarrow \{',+,X,U\}$

```

Generation: 20
Test:
X X X X X U X X      U X + X U      X X U X X X X X
X X X X X      + U U U U + U U U + X X X X X X
X X X      U X U +      X +      + X      + U X U X X X
X X      U X X X U +      +      X      +      U X X X U X X
X      U X      U      U      U      +      U      U      X U X
U X X X      X X X U X X X U X X X U X X X X X X U
X      U X U X X X      U      U      X U X U X U X
X + + U X X X      X X      X X      X X X U + + X
      U      + U U      U X U      U X U      U U +      U
      U X      X      U X U + X + U X U      X      X U
U U + + U X U X X U X X X U X X U X U + + U U
X + + U X + U      X U + X U X U X + U X      X U + +
X U      X      X U + X U X U X + U X      X      U X
U U + + U X U X X U X U X      U X X U X U + + U U
      U X      X      U X U + X + U X U      X      X U
      U      + U U      U X U      U X U      U U +      U
X + + U X X X      X X      X X      X X X U + + X
X      U X U X U X      U      U      X U X U X U X
U X X X      X X X U X X X U X X X U X X X X X U
X      U X U X U X      U      U      X U X U X U X
X + + U X X X      X X      X X      X X X U + + X
      U      + U U      U X U      U X U      U U +      U
      U X      X      U X U + X + U X U      X      X U
U U + + U X U X X U X X X U X X U X U + + U U
X U      X      X U + X U X U X + U X      X      U X
+ + U X + U      X U X      X U X      U + X U + +
X U      X      X U + X U X U X + U X      X      U X
U U + + U X U X X U X U X      U X X U X U + + U U
      U X      X      U X U + X + U X U      X      X U
      U      + U U      U X U      U X U      U U +      U
X + + U X X X      X X      X X      X X X U + + X
X      U X U X U X      U      U      X U X U X U X
U X X X      X X X U X X X U X X X X X X X U
X      U X      U      U      U      +      U      U      X U
X X      U X X X U +      +      X      +      + U X X X U X X
X X X      U X U +      X +      U      + X      + U X U X X X
X X X X      X      + U U U U + U U U U +      X      X X X X
X X X X X U X X      U X + X U      X X U X X X X X
GroundTruth:
X X X X X U X X      U X + X U      X X U X X X X X
X X X X X      + U U U U + U U U U +      X      X X X X
X X X      U X U +      X +      U      + X      + U X U X X X
X X      U X X X U +      +      X      +      + U X X X U X X
X      U X      U      U      U      +      U      U      X U X
U X X X      X X X U X X X U X X X U X X X X X X U
X      U X U X U X      U      U      X U X U X U X
X + + U X X X      X X      X X      X X X U + + X
      U      + U U      U X U      U X U      U U +      U
      U X      X      U X U + X + U X U      X      X U
U U + + U X U X X U X X X U X X U X U + + U U
X U      X      X U + X U X U X + U X      X      U X
+ + U X + U      X U X      X U X      U + X U + +
X U      X      X U + X U X U X + U X      X      U X
U U + + U X U X X U X U X      U X X U X U + + U U
      U X      X      U X U + X + U X U      X      X U
      U      + U U      U X U      U X U      U U +      U
X + + U X X X      X X      X X      X X X U + + X
X      U X U X U X      U      U      X U X U X U X
U X X X      X X X U X X X U X X X X X X X U
X      U X      U      U      U      +      U      U      X U
X X      U X X X U +      +      X      +      + U X X X U X X
X X X      U X U +      X +      U      + X      + U X U X X X
X X X X      X      + U U U U + U U U U +      X      X X X X
X X X X X U X X      U X + X U      X X U X X X X X
Generation: 20, PASSED

Generations Passed: 20/20

```

C++ runs

ca_module_tb2.cc:


We want to indicate that the CA module works for 10 TESTS. We don't confuse now GENERATIONS with TESTS.


In every test there is a new random seed, which is produced by `generate_random_lut()` as usual.

There is no need to show a lot of generations of the CA. Only one will do the job (gen:0 and gen:1)! That's because we want to show that the computation of next gen is doable.

Another difference is that, in the CMD .exe file, the TestCA's outcome is next to RunCA's outcome at the right place (horizontally), instead of being under it (vertically).

On the right we can see the generation 1 of test 10.

```
Generation: 1  
Groundtruth:  
  
Test: 10, PASSED  
Tests Passed: 10/10
```

```
Test Table:  

```

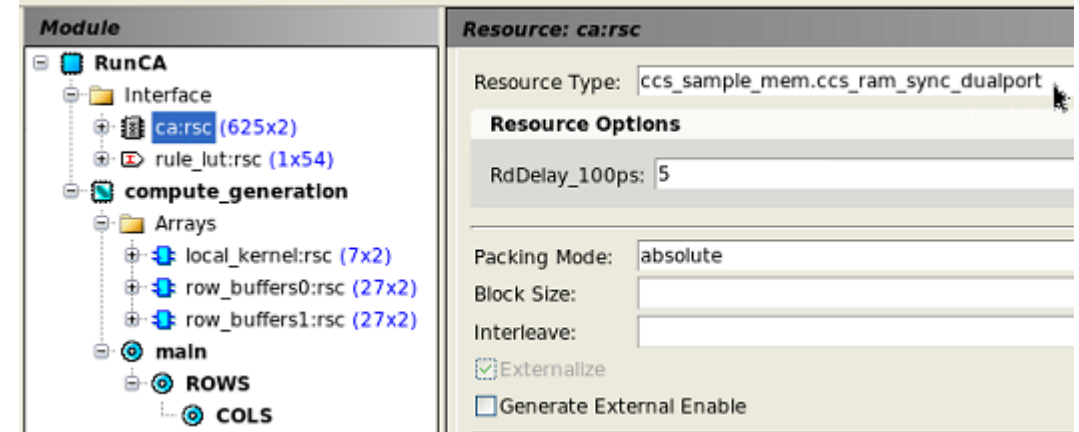
Catapult - Version 1 & 2

For the runs, we selected grid dimensions **N=25**, **M=25**. The values are small enough for the results to be manageable.

Starting from version 1 we progressively add options. Clock speed is set to **250 MHz**, and ca grid is stored in a **dual port memory**.

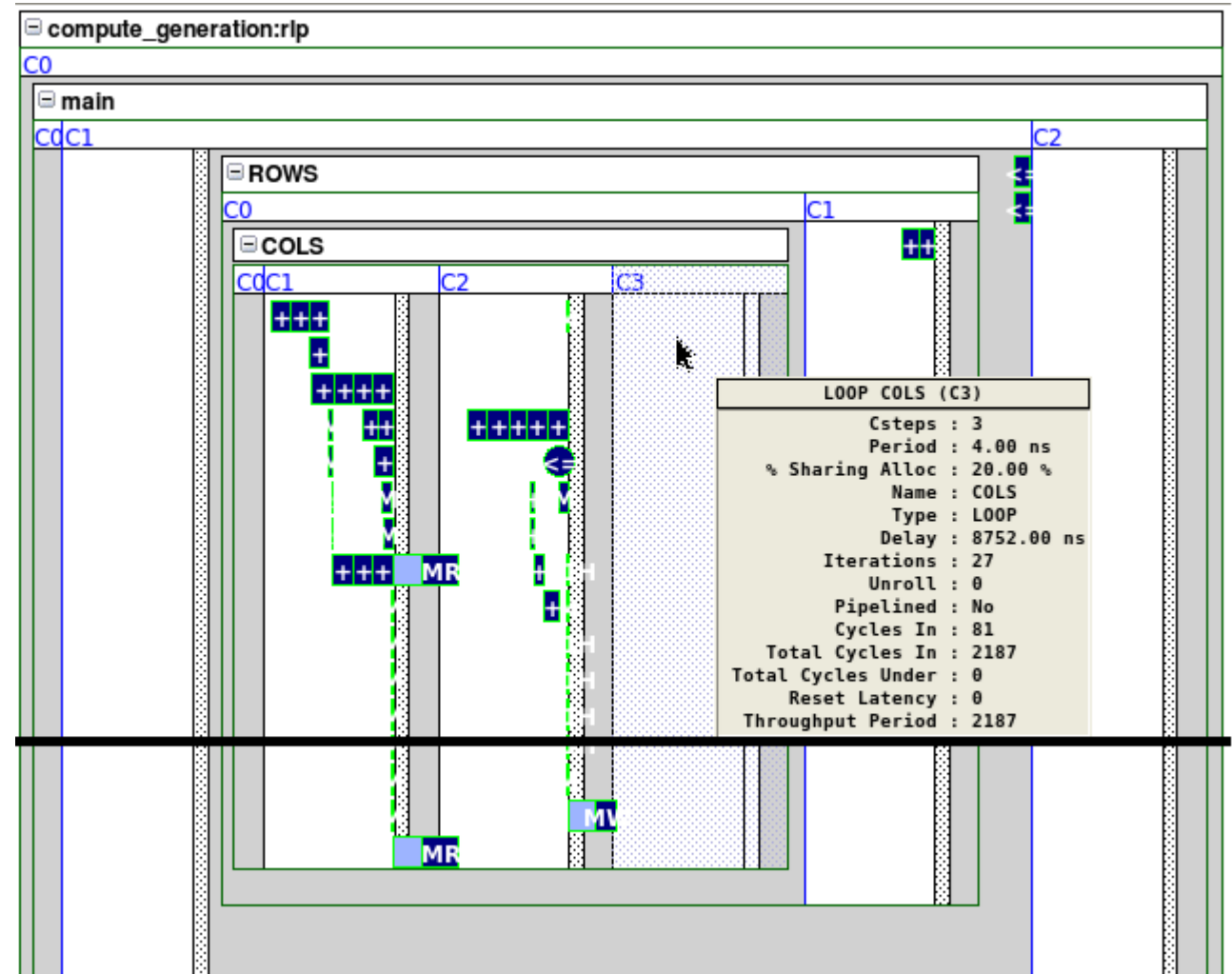
Row **buffers** are stored independently to reduce IO dependencies (if they are in memory). Here they are stored on **registers** (we will show later an alternative). **local_kernel** is on **registers** and **rule_lut** comes with **wires**.



Version 2 has the extra option of Effort Level: high and Design Goal: latency



Catapult - Version 1 & 2

The schedule of version 2 and the table of the two runs:

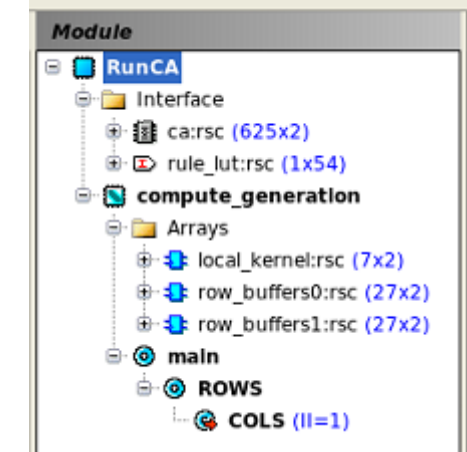


Solution /	Latency Cycles	Latency Time	Throughput Cycles	Throughput Time	Total Area	Slack
 RunCA.v1 (extract)	2940	11760.00	2945	11780.00	2073.97	0.64
 RunCA.v2 (extract)	2211	8844.00	2216	8864.00	1999.77	0.80

Catapult - Version 3 & 4

Next step is to perform pipeline at COLS loop with $II=1$.
The compilation fails and shows the below message:

#	Message
i	Select qualified components for data operations ...
i	Apply resource constraints on data operations ...
x	Feedback path is too long to schedule design with current pipeline and clock constraints.
x	Schedule failed, sequential delay violated. List of sequential operations and dependencies:
x	MEMORYREAD "COLS:if:read_mem(ca:rsc.*)" ca_v6_enhanced.h(74,50,28)
x	<u>MEMORYWRITE "COLS:if#1:write mem(ca:rsc.*)" ca_v6_enhanced.h(81,12,12)</u>
x	Feedback path is too long to schedule design with current pipeline and clock constraints.
#	ca_v6_enhanced.h(74,50,28): chained data dependency at time 22cy+3.2
#	from operation MEMORYREAD "COLS:if:read_mem(ca:rsc.*)" with delay 0.5

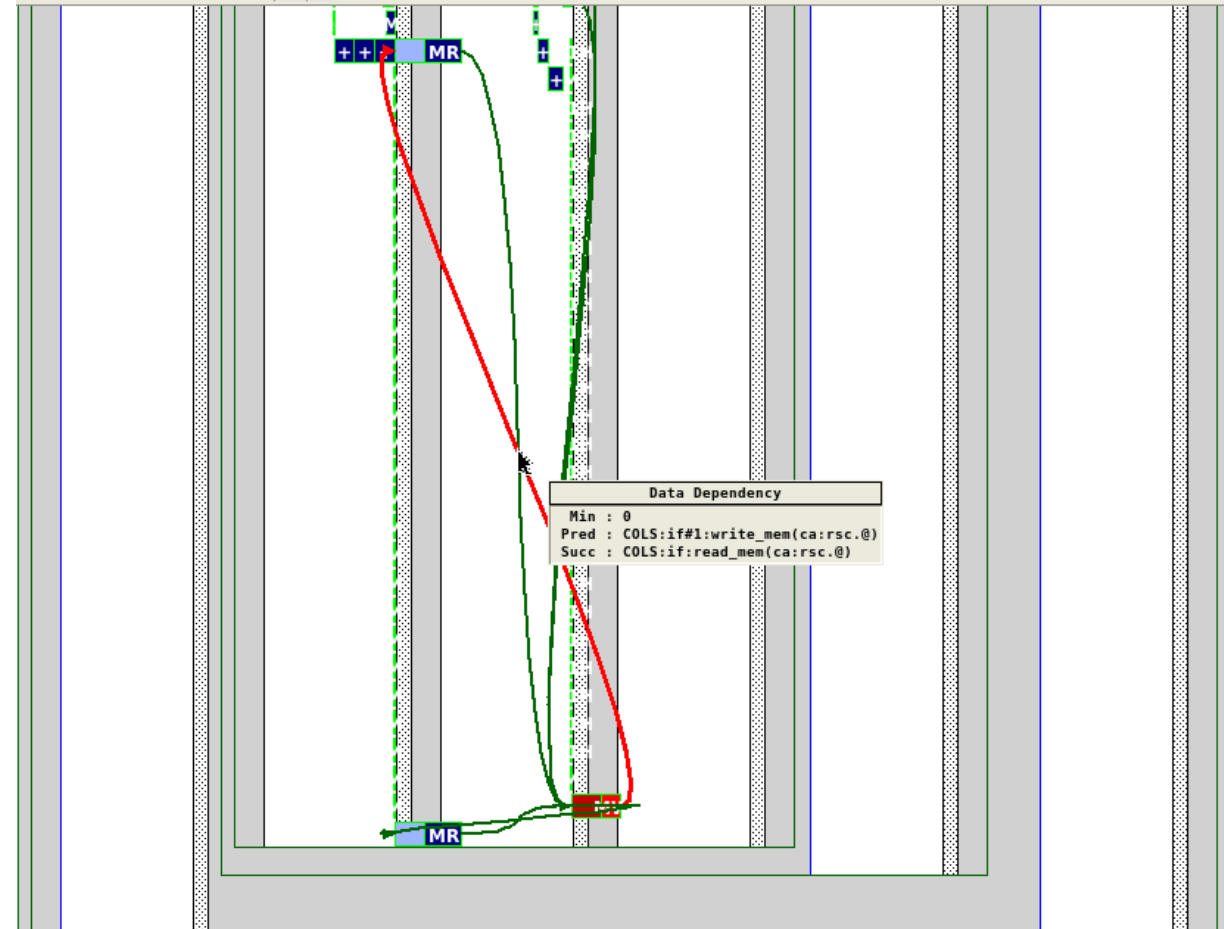


Catapult - Version 3 & 4

As we can see from the schedule of version 2, catapult assumes a dependency. Examining the algorithm, this dependency is not real and offers no functionality.

We set catapult to ignore this dependency with the command below and reschedule with the existing options.

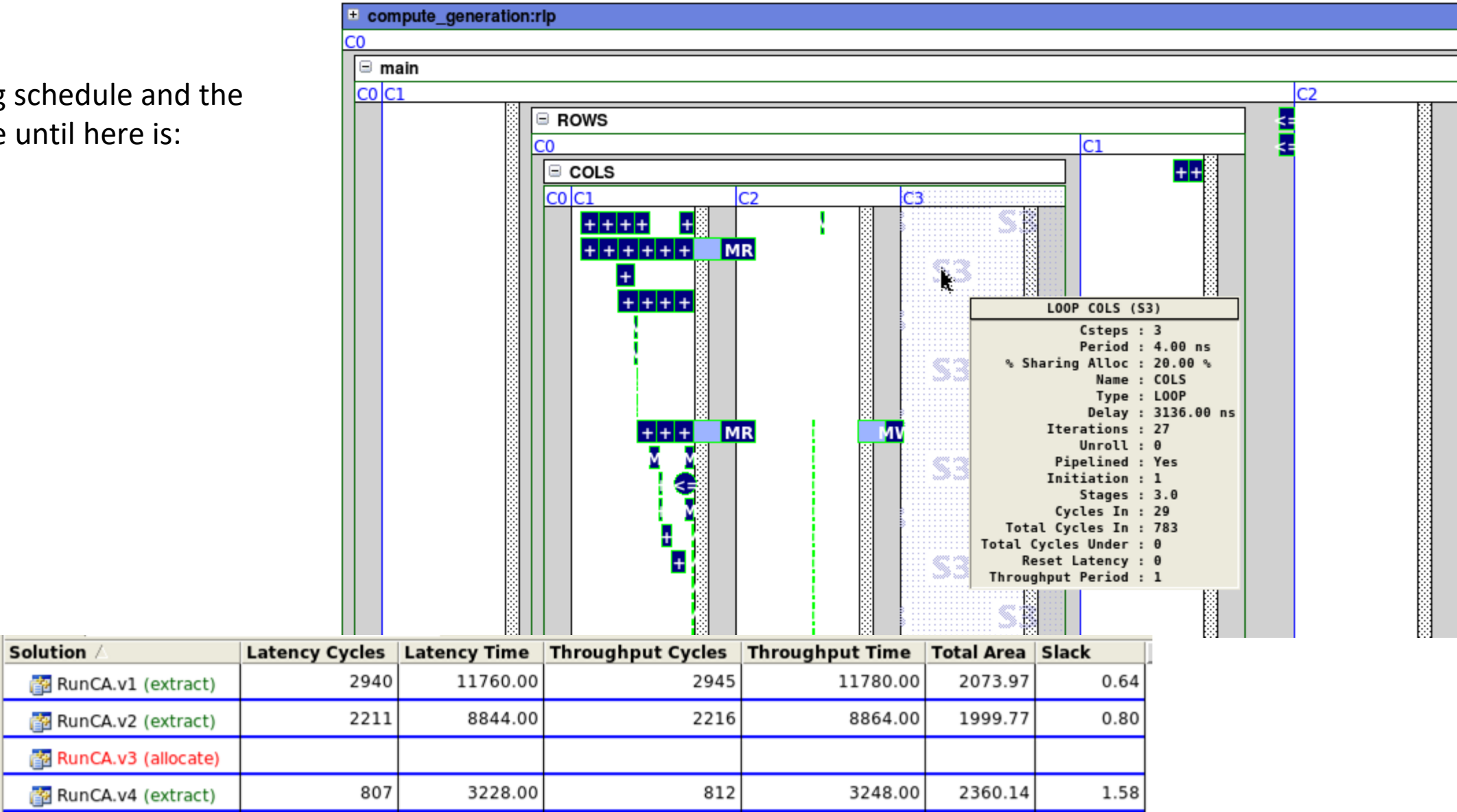
*the command ignores the dependency shown in the schedule not the one that the error was initially produced.



```
✖ Design 'RunCA' could not schedule partition '/RunCA/compute_generation' - could not schedule even with unlimited resources
📄 ignore_memory_precedences -from COLS:if#1:write_mem(ca:rsc.@) -to {COLS:if:read_mem(ca:rsc.@) COLS:else:read_mem(ca:rsc.@)}
ℹ Branching solution 'RunCA.v4' at state 'architect'
# Makefile for Original Design + Testbench written to file './scverify/Verify_orig_cxx_osc_i.mk'
ℹ CDesignChecker Shell script written to '/home/user3/georbaba4/final_project/hls_files/Catapult/RunCA.v4/CDesignChecker/design_checker.sh'
# /RunCA/compute_generation/compute_generation:rlp/main/ROWS/COLS/COLS:if:read_mem(ca:rsc.@)/IGNORE_DEPENDENCY_FROM COLS:if#1:write_mem(ca:rsc.@)
# /RunCA/compute_generation/compute_generation:rlp/main/ROWS/COLS/COLS:else:read_mem(ca:rsc.@)/IGNORE_DEPENDENCY_FROM COLS:if#1:write_mem(ca:rsc.@)
📄 go allocate
```

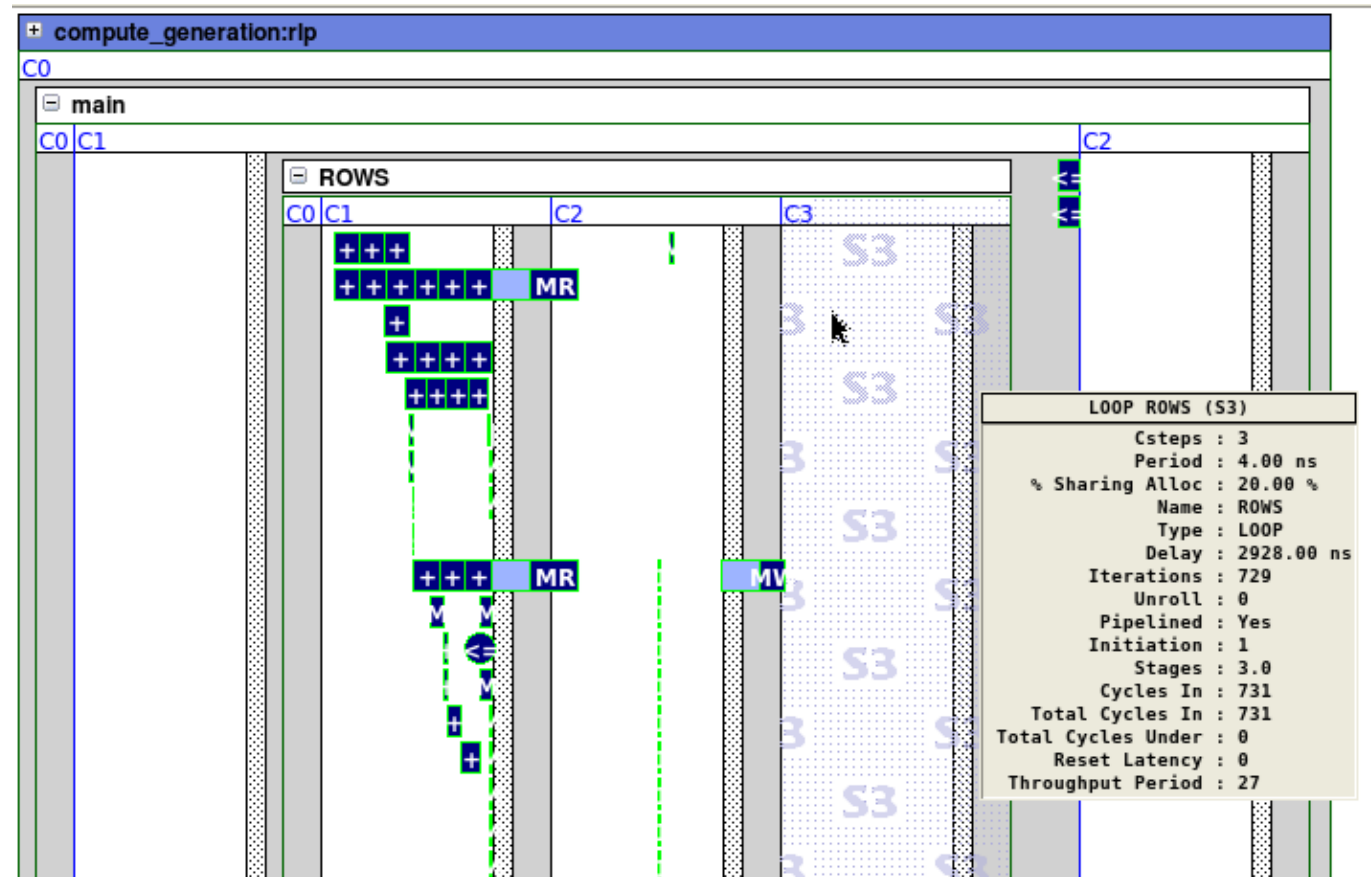
Catapult - Version 3 & 4

The resulting schedule and the metrics table until here is:









Catapult - Version 5 & 6

Since buffers are stored on registers we can continue and pipeline ROWS loop as well, with $II=1$. The schedule is:



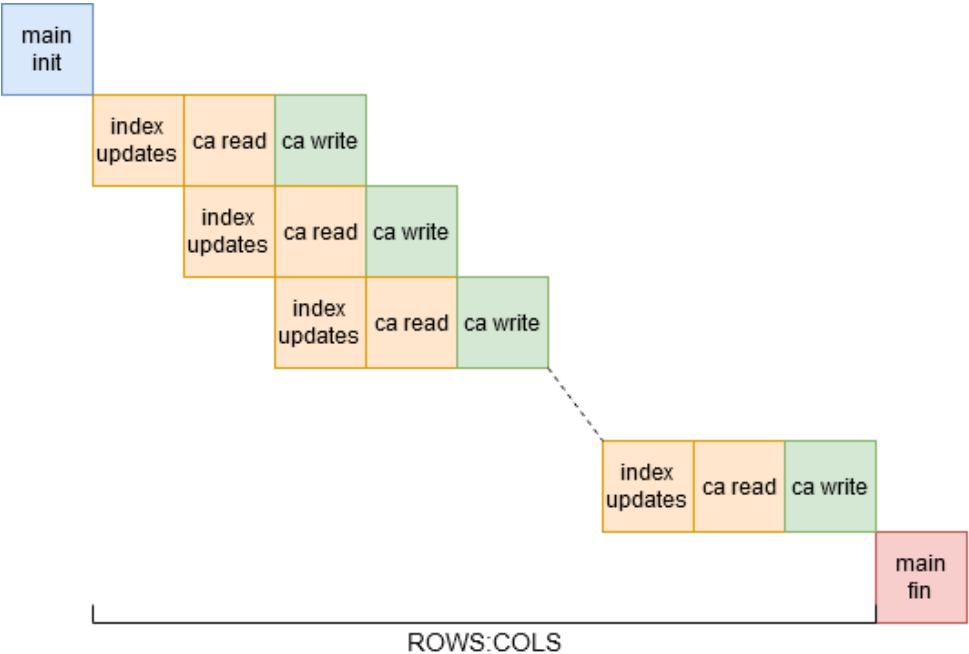
Catapult - Version 5 & 6

For the final version, v6, we subtract the slack from the clock period and round it up. We end up using a 400 MHz clock. The resulting metrics table is:

Solution /	Latency Cycles	Latency Time	Throughput Cycles	Throughput Time	Total Area	Slack
 RunCA.v1 (extract)	2940	11760.00	2945	11780.00	2073.97	0.64
 RunCA.v2 (extract)	2211	8844.00	2216	8864.00	1999.77	0.80
 RunCA.v3 (allocate)						
 RunCA.v4 (extract)	807	3228.00	812	3248.00	2360.14	1.58
 RunCA.v5 (extract)	729	2916.00	733	2932.00	2409.37	1.58
 RunCA.v6 (extract)	729	1822.50	733	1832.50	2409.37	0.08

Catapult - Version 5 & 6

Throughput cycles can be calculated as follows:



Throughput cycles

@main: 1 cycle in the beggining and 1 in the end = 2 cycles => Total = 731 + 2 = 733 cycles

@ROWS:COLS (L = 3, ll = 1, iter = 27*27 = 729) : 3 + 1*728 = 731 cycles

 RunCA.v6 (extract)	729	1822.50	733	1832.50	2409.37	0.08
--	-----	---------	-----	---------	---------	------

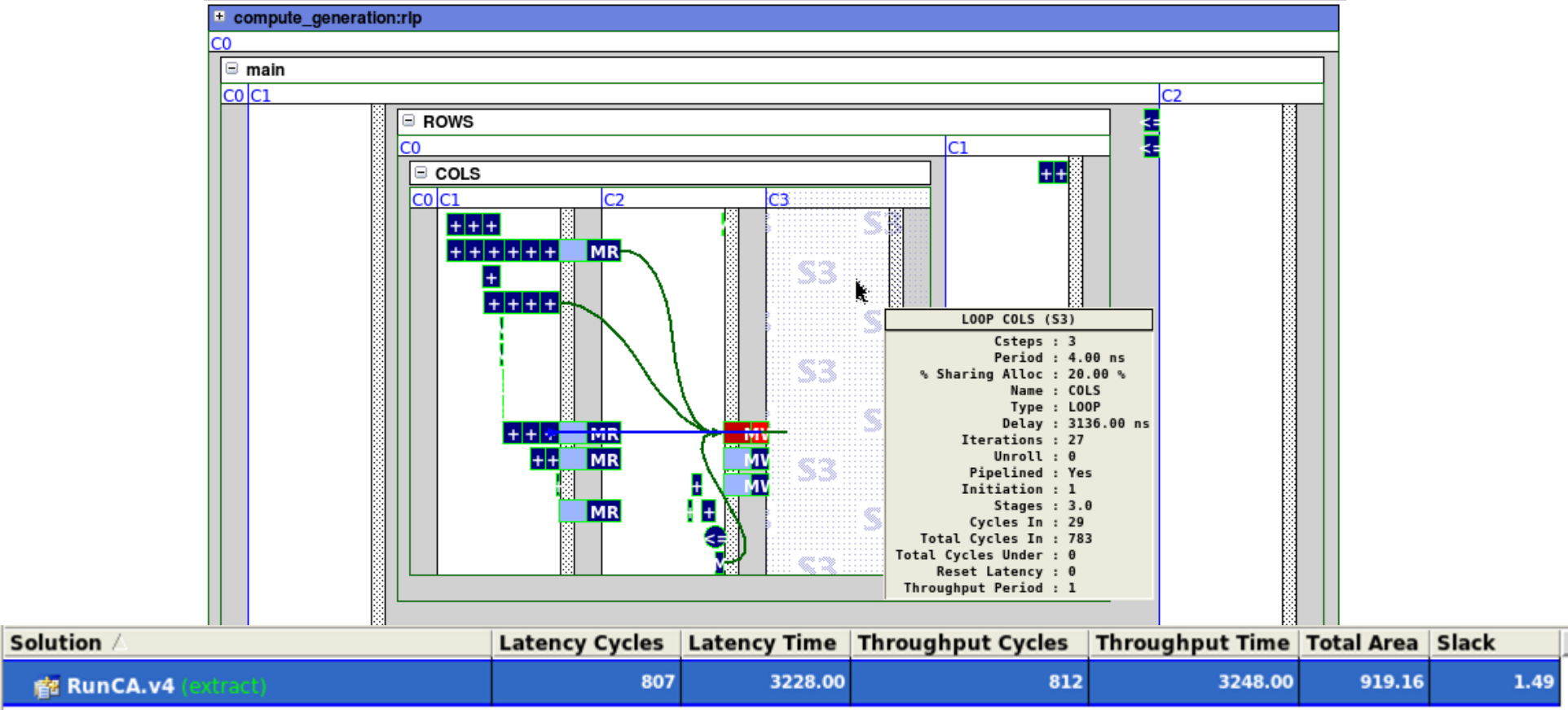
Questasim results

Produced RTL seems to have the expected functionality, as for tests provided and Questasim.

```
VSIM 2> run -all
# SCVerify intercepting C++ function 'RunCA::compute_generation' for RTL block 'RunCA'
# Info: HW reset: TLS_rst active @ 0 s
# Generations Passed: 200/200
# Info: Execution of user-supplied C++ testbench 'main()' has completed with exit code = 0
#
# Info: Collecting data completed
#   captured 200 values of ca_IN
#   captured 200 values of ca
#   captured 200 values of rule_lut
# Info: scverify_top/user_tb: Simulation completed
#
# Checking results
# 'ca'
#   capture count      = 200
#   comparison count   = 200
#   ignore count       = 0
#   error count        = 0
#   stuck in dut fifo  = 0
#   stuck in golden fifo = 0
#
# Info: scverify_top/user_tb: Simulation PASSED @ 366504250 ps
# ** Note: (vsim-6574) SystemC simulation stopped by user.
# 1
#
VSIM 3>
```


Catapult - Version extra





We can store buffers on dual port memory, if they require big enough size. The drawback would be that we can only pipeline COLS loop. With this option the Total area is greatly reduced without much of a setback on throughput . Also, slack optimization could be performed. The schedule and the metrics for this option is:



Final thoughts

- Through optimization achieved 75.1% decrease on throughput cycles, compared to the initial design, leading to better performance.
- Row buffers can be stored either on registers or memory with small throughput loss and great area gains.
- If the problem size is big enough then rule_lut could be stored on registers or small memory instead of just wire input.
- For a grid with dimensions, for example, 400x400, the final throughput would be one result every 391442.64 nsec or 391.442 μ sec, which seems sensible.

Extended results shown below for buffers on memory.

Solution /	Latency Cycles	Latency Time	Throughput Cycles	Throughput Time	Total Area	Slack	
 RunCA.v4 (extract)	807	3228.00	812	3248.00	919.16	1.49	250 MHz 25x25
 RunCA.v5 (extract)	807	2065.92	812	2078.72	919.16	0.05	390 MHz 25x25
 RunCA.v6 (extract)	162807	416785.92	162812	416798.72	1399.23	0.43	390 MHz 400x400
 RunCA.v7 (extract)	162807	361431.54	162812	361442.64	1401.41	0.09	450 MHz 400x400

References

HLS course slides

12_Schedule Failure Debug.pdf

[Two-Dimensional Cellular Automata](#)
[cppreference.com](#)

